instead of = to emphasize that consideration must be taken of the quantization on $r$. Sample results from a computer program to compute $S_k$ in this manner are given in Figures 2 and 3.

Consideration of the templates $S_k$ reveals much about the behavior of the Hough line detector in noisy situations and helps to establish threshold settings. Figure 2 (previous page) shows that the template for line (120°, 5) integrates 21 spatial cell outputs. For a binary 21 × 21 image, as in Figure 2, perhaps a threshold of 15 would be appropriate for accumulator cell (120°, 5), depending on the noise situation. It has been assumed that the line extends off the retina in both directions. If shorter line segments are to be detected, the threshold must be suitably lowered. Figure 3 shows rather sloppy templates for a set of three parallel lines at a coarse quantification on $r$. Since there are only seven levels of $r$, the templates for all 30° lines are evident. Note that a few 120° lines would stimulate many cells of each template and could be confused with a thin 30° line. A blob oriented along a 30° line could cause similar confusion.


## Summary

It is clear that the Hough transformation can be implemented in hardware in a simpler manner than described in Hough's original patent [2]. Examination of the set of spatial cells being integrated by a single cell of the accumulator array shows that a simple template match is performed. Awareness of this fact is not evident in the literature, probably because sequential software implementations, for efficiency, have used the "distribution" approach. Consideration of the templates, i.e. the sets of cells being integrated in the accumulator cells, is important even in the sequential implementation in order to set thresholds and understand detector response to the possible input curves under given noise situations. Although only straight line detection was discussed, detection of any general curve is possible provided that a mapping can be established from each spatial point $(x, y)$ to the set of all cells $P = (\alpha_1, \ldots, \alpha_n)$ in the Hough retina which parameterize curves passing through $(x, y)$.



Received October 1975; revised January 1976



**References**

1. Duda, R.O., and Hart, P.E. Use of the Hough transformation to detect lines and curves in pictures. *Comm. ACM 15*, 1 (Jan. 1972), 11-15.
2. Hough, P.V.C. Methods and means for recognizing complex patterns. U.S. Pat. 3069654, Washington, D.C., Dec. 1962.
3. Kimme, C., Ballard, D., and Sklansky, J. Finding circles by an array of accumulators. *Comm. ACM 18*, 2 (Feb. 1975), 120-122.
4. Merlin, P., and Farber, D. A parallel mechanism for detecting curves in pictures. *IEEE Trans. Computers C-24*, 1 (Jan. 1975), 96-98.
5. Perkins, W.A., and Binford, T.O. A corner finder for visual feedback. *Computer Graphics and Image Processing 2*, 3/4, (Dec. 1973), 355-376.
6. Rosenfeld, A. *Picture Processing by Computer*. Academic Press, New York, 1969.




Short Communications
Programming Languages

# What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?


Niklaus Wirth
Federal Institute of Technology (ETH), Zürich, and
Xerox Palo Alto Research Center



**Key Words and Phrases: syntactic description language, extended BNF**
**CR Categories: 4.20**


The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietory circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasymbols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.




Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

4. It avoids the use of an explicit symbol for the empty string (such as ⟨empty⟩ or ϵ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax     = {production}.
production = identifier "=" expression ".".
expression = term {"|" term}.
term       = factor {factor}.
factor     = identifier | literal | "(" expression ")" |
             "[" expression "]" | "{" expression "}".
literal    = """" character {character} """".
```

Repetition is denoted by curly brackets, i.e. {a} stands for ϵ | a | aa | aaa | . . . . Optionality is expressed by square brackets, i.e. [a] stands for a | ϵ. Parentheses merely serve for grouping, e.g. (a|b)c stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

Received January 1977; revised February 1977

Short Communications
Programming Techniques

# A Note On Reflection-Free Permutation Enumeration

Mohit Kumar Roy
Jadavpur University, India

**Key Words and Phrases: permutations, reflection-free generation**
**CR Categories: 5.30**

Earlier it was shown by the present author [1] that among the classical algorithms for the generation of permutation sequences, only the Trotter-Johnson algorithm [2, 3] has the property that the reflection of any permutation in the first half of the enumeration appears only in the second half. Two permutations are called reflections of each other if one read from left to right is the same as the other read from right to left. Lenstra [4] has discussed the usefulness of this property in certain applications. Recently Ives [5] has produced a series of four permutation algorithms of which two (algorithms *c* and *d*) possess the said property.

To prove that the set of first $n!/2$ arrangements generated by algorithm *c* is reflection-free, let the $n!$ permutations of $n$ marks be numbered serially from 0 to $n! - 1$ in the order they are produced. Also let us divide the $n!$ arrangements generated by the algorithm into $(n - 2)!$ consecutive groups of $n(n - 1)$ arrangements each. The first arrangement of each group will have the first and the $n$th marks in their original positions [5]. The other positions will have the permutations of the marks $2, 3, \ldots, (n - 1)$ in the order of their generations by the same algorithm. Let $N^0_{n-2}$ and $N^1_{n-2}$ be the serial numbers of two permutations of the marks $2, 3, \ldots, (n - 1)$ such that they are reflections of each other. Having taken into consideration the procedure by which the permutations in each group are generated, it is easy to show that the reflecitons of each of the permutations in the group corresponding to $N^0_{n-2}$ will appear in the group corresponding to $N^1_{n-2}$. Therefore, if the first $(n - 2)!/2$ generations of $(n - 2)$ marks are reflection-free, the first $n!/2$ generations of $n$ marks will also be reflection-free. Verification when $n = 2$ and 3, proves by induction the reflection-free property of algorithm *c*. Since algorithm *d* differs from *c* only in the direction in which the first mark is passed through the other marks, the above arguments hold good in case of algorithm *d* too.

Finally, following Ives [5], we may say that of the three permutation algorithms having the said property possibly algorithm *c* is the most efficient.

Received August 1976; revised February 1977

**References**

1. Roy, M.K. Reflection-free permutations, rosary permutations and adjacent transposition algorithms. *Comm. ACM 16*, 5 (May 1973), 312-313.
2. Trotter, H.F. Algorithm 115: Perm. *Comm. ACM 5*, 8 (Aug. 1962), 434-435.
3. Johnson, S.M. Generation of permutations by adjacent transposition. *Math. Comput. 17* (1963), 282-285.
4. Lenstra, J.K. Recursive algorithms for enumerating subsets, lattice-points, combinations and permutations. Rep. BW 28/73, Mathematisch Centrum, Amsterdam, 1973, p. 38.
5. Ives, F.M. Permutation enumeration: Four new permutation algorithms. *Comm. ACM 19*, 2 (Feb. 1976), 68-72.



Author's address: Regional Computer Centre, Calcutta, Jadavpur University Campus, Calcutta F00032, India.